Faceted Values and Information Flow Control in the Wild

Konstantinos Xirogiannopoulos Benjamin Bengfort
University of Maryland, College Park
{*kostasx,bengfort*}@cs.umd.edu

Abstract

It has become increasingly popular to create rich web applications through the plug-and-play integration of web services by embedding publisher specific JavaScript code into an application's native code base. Typically, a plug-and-play workflow relies on the inclusion of snippets of third party code in the client application, customized with confidential API keys or site-specifc information. Modern browsers provide techniques like sandboxing and cross-domain exclusion attempt to protect confidential information from cross site scripting (XSS) and injection attacks. However, once third party code is embedded, it is treated as a native part of the code base and is executed with the same privileges. As a result, this *"outsider code"* is a vector for malicious behavior and the leakage of confidential information through a variety of channels within the code itself.

Understanding the *information flow control* of programs is a systematic approach to determining how information (data) stored in a program's variables and gets propagated throughout the code. Information flow control techniques detect the way data is passed around by specifying a security level for each variable and provide monitoring to prevent the propagation (flow) of sensitive information to public observers. In this survey paper, we will expand upon the basic concepts and primary contributions of information flow control and enumerate many of the existing techniques proposed by the academic community. We will then turn to focus on a specific technique: *dynamic information flow* control using faceted values, developed by Austin and Flanagan. We have implemented the faceted evaluation semantics for a simpler version of the λ^{facet} language proposed in the paper; based on the F language. We called this language F^{facet} and have implemented its big step evaluation and faceted evaluation semantics in *OCaml*.

1. Summary

The software systems we use everyday process and propagate many types of information, some of which is highly confidential and should be treated specially to ensure privacy. Programs require the ability to use confidential data, from social media passwords passwords for universal login to bank information and credit card numbers for online shipping. As this information is used to conduct the business of the user, data is propagated through the program via assignment, reads and writes, and branching. Moreover, as software systems increase in complexity, the propagation path of secure information tends to get longer, making debugging very difficult in the case of leaks.

The focus of security related research is on securing the communication of confidential data between programs with techniques such as cryptography, access control, and firewalls. Generally, sending an encrypted message between machines or processes is considered secure. However, any sort of "security" guarantees we claim about the message are rendered invalid when the message is decrypted into plain text on the receiver because it is unusable in its encrypted form. A new set of issues must be reasoned about *what* or *who* has access to information as it *flows* through the system in order to serve its purpose. On the one hand, it is usually the case that programs which receive encrypted messages execute in isolation and therefore, unless already compromised, have a low probability of malicious observers viewing confidential information. This expectation requires a secure code base which is run completely in a controlled environment like a server. However, web applications distribute and execute code in the browser, opening up many channels of information leakage.

Therefore the focus of this paper is the case where code is publicly available and whose execution can be observed alongside the code.

Data is *propagated* through variables inside a program through assignment, and the chain of assignments form a path that information flows through during execution. Values often flow through complex paths in modern programs, as many different variables are required to hold the same information at different times in the execution cycle, for example in passing variables as arguments to functions, creating closures, or preventing scoping issues. A standard execution utilizes values of different confidentiality levels simultaneously: public values, the normal pieces of interaction that require no security guarantee, and private values that should be used specifically for secure transactions and never revealed. *Information flow theory* studies the way both public and private values travel from variable to variable, in order to identify the ways in which information can *leak* to observers of data propagation.

For the purposes of the cases we have studied, there are only two security levels values can have, high and low confidentiality. High confidentiality, H, includes data that must be maintained as secret, usually API keys, password, or personally identifying information. Low confidentiality, L specifies data that may be observed publicly without repercussion, and is the most common type of data in a system. In terms of the level of confidentiality, $L < H$. The primary purpose of work done in this area focuses on techniques to *detect* and *avoid* unwanted information flows automatically with the idea that if programmers can detect information flows, they can take action to rewrite code so the flows don't occur. Additionally, there are also techniques for automatically eliminating or avoiding unwanted information flows at run time, although this incurs a performance or early termination costs.

The rest of the paper is organized as follows: we begin with a survey that includes background and fundamentals on a variety of information flows which we use to then outline a variety of information flow control mechanisms. Our focus then shifts to a specific mechanism, faceted evaluation, and the features and benefits of this technique. We conclude by evaluating the approach and discussing challenges and future work.

2. Information Flow Control

Broadly, our research problem is to investigate techniques to prevent unwanted leaks of confidential information while information flows through a program. There are primarily two main classes of unwanted leaks, *explicit* and *implicit* [14, 15]. Explicit flows are caused by the *direct assignment* of a confidential value stored inside an H variable, `h` being assigned to an L variable, `l` – $l := h$. This type of assignment is relatively straight forward to detect both with static analysis and dynamic analysis at runtime, which we will discuss later in Section 2. However, the more complicated, and more nefarious information flows are implicit, observed through side channels in the program execution.

Side Channels and Implicit Flow

Aside from explicit assignment, the means through which information flow can leak implicitly (e.g. inferred from some property of the execution) are commonly known as *side channels*. Side channels include monitoring power consumption [13] or other resources like CPU and memory [16]. Most of the program analysis literature focuses on controlling a side channel where information is inferred through the *control flow* of a program, usually called *implicit flows*. Implicit flows are significantly more difficult to pinpoint than explicit flows because the sensitive data is never assigned directly to an L variable. Instead, these flows occur through the inspection of the value of L variable that might be changed based on a branch in the execution of the program because of a H variable. The motivating example of this can be seen in Figure **??**.

The ultimate goal of language-based techniques (as well as many other methods for detecting implicit flows) is the security guarantee of *non-interference*. Non-interference states that *any* alteration or execution with values of the H confidentiality level should not affect the end result **or** the program execution in any way that might expose their value to potentially malicious, public observers. Another way to state this is that *no data visible to other principals is affected by confidential data* meaning that H variables must not interfere with L values that are publicly observable. Non-interference can be naturally expressed by semantic models of program execution, however this policy is very strict and enforcing it in a practical way for real execution contexts is difficult, as expressed by Dorothy Denning:

> The primary difficulty with guaranteeing security lies in detecting and monitoring all flow causing operations. This is because all such operations in a program are not explicitly specified or indeed even executed! – Dorothy Denning

Early research, such as the seminal paper by Dorothy E. Denning, modeled information flow as a *lattice* [8]. Lattice models are useful in that they embed secure information flow static inspection can reveal both explicit and implicit flows. In a lattice, information is said to flow from a security class A to a security class B if any information associated with security class A *affects* the value of information associated with security class B. The lattice itself is a structure consisting of a finite, partially ordered set of security classes and their values that maintains the least upper bound and greatest lower bound operators of the set. Although we're studying simply two security classes, H and L, the lattice model can be used to reason about and detect flows between a higher number of finite classes.

Based on this primary theory of information flow, our primary research question that we have surveyed concerns techniques for *enforcing* security policies in both a *static* and *dynamic* context. In order to enforce any information flow policy there must be a method for *detecting* problematic flows, which are known as *information flow control* mechanisms. For the purpose of this survey we have focused on *language-based* methods since they are most relevant to the paper our survey was inspired by.

Static Analysis of Information Flow

The first category of techniques that have been developed to detect problematic flows involves *static information flow control* that utilizes static analysis before a program executes. The most common technique in this category is a security-type system. As in a standard type system, a security-type system is a set of semantic rules that assign a *type* to a construct in a program (variables, expressions, functions, etc). However, instead of assigning a type that describes the kind of data the expression conveys like *integer* or *string*, it instead assigns a label based on the security level of the expression – it's *security-type*. These systems are so similar to standard type systems that well developed features of type systems have been successfully and usefully integrated into security-type systems.

Specifically, security-type systems like the one described in [14] assign a security-type to each expression in the language. Security type violations occur when an expression modifies or assigns another expression of a different security type. By inspecting the type of a variable, a static analysis can automatically detect problematic explicit flows by not allowing the assignment of one security type to a higher one. Implicit flows can also be detected by determining the types of all expressions in a control block, and by determining if lower security types are being affected by conditions of the higher types. Figure **??** shows an example of a simple security-type system.

Like standard type systems, security-types are strict and are necessarily conservative in order to detect flows. For this reason, they reject a reasonable amount of secure programs. Moreover, it is assumed that the confidentiality labels of the variables are known statically before run time. This is a strict condition however in a computing environment where elements are interdependent and may change dynamically. Static flow control methods therefore prove a poor fit for larger dynamic environments where each variable cannot hold both static data and a static security label. In the situation where security labels need to be added dynamically at runtime, we must turn to *dynamic information flow control*.

Dynamic Analysis of Information Flow

The fundamental difficulty of the dynamic enforcement of security labels (and therefore the primary difficulty in dynamic information flow control) is that there is no way to inspect all paths of a program's execution simultaneously. At runtime, only the labels for a single path can be known – the current execution path.

Dynamic information flow control techniques therefore have difficulty when it comes to implicit flows. When the execution branches through conditional statements into multiple paths, the possibility of an implicit flow is introduced where one path utilizes a private H variable, in such a way modifies or alters a public L variable. In order to maintain the invariant of non-interference, these techniques simply choose to get *stuck* and completely stop execution when such a branch occurs.

Previous work has defined execution semantics that dynamically add security labels and track information on the fly. Techniques such as *no-sensitive upgrade semantics* [3, 17] entirely halt the execution as soon as a branch is detected where public values are updated in a conditional statement depending on confidential variables. Slightly more relaxed are *permissive-upgrade* semantics [4] which accept some programs with potential implicit flows by marking the execution as having lost track of the correct public value inside the conditional branch. Both of these semantics essentially "throw away" any public values to ensure that they aren't affected by the branch that caused the implicit flow. Nevertheless, both of these semantics can still get stuck when they reach a case where the execution is dependent on a thrown-away public value.

Both of these semantics reamin *sound* by maintaining a relaxation of the non-interference guarantee called *termination-insensitive non-interference (TINI)*, which we will elaborate upon in Section 4. The terminiation-insensitivity excludes the possibility of getting stuck (early termination) and, though sound, may still reject many secure programs that include implicit flows which cannot be detected outright by the semantics. These problems make control mechanisms particularly difficult to use in dynamic languages like JavaScript.

3. Faceted Values for Dynamic Information Flow Control

There also exist a distinct set of techniques that allow for dealing with the problem of implicit flows, without necessarily actually "detecting" places in the code where they may occur. These techniques take a completely different track into solving this problem, by eliminating it completely without the need for anything to be changed within the code itself. A few of these techniques include *shadow execution* and *secure multi-execution* [7, 9]. These techniques both work by essentially executing *two* distinct copies of a program; a public copy and a private copy, simultaneously and independently from each other. The public does not have access to any confidential values, and is the only one that handles requests and interacts with all public observers that have no access to private values. The private copy on the other hand maintains and handles all confidential values, never sends any information over the network.

Both of the above techniques eliminate the problem of potential leaks of confidential values to public observers, without the need to detect and fix any sort of problematic flows within the code itself. The main trade-off here is that these techniques introduce a large amount of overhead since we need to simultaneously execute two copies of the program concurrently. This becomes increasingly more expensive as the number of *principles* in the program increases. Principles are simply the data values that need to be obscured, and for which this dual execution actually happens.

The paper we have based this survey on is called **Multiple Facets for Dynamic Information Flow** [5] and it's main contribution is that it proposes a sound alternative to the above techniques, which is substantially more efficient than executing dual copies of the program. This paper proposes a method called *Faceted Evaluation*, which essentially maintains dual *values* for specific variables, and alter the value that is used for computation depending on the individual observer's *view* of the values. These values are called *Faceted Values*, and their intuition is quite simple. There exist the idea of *principles* within the code, in the sense that each principle defines the "domain" of observers that is meant to see the public value. Each observer is characterized by a *view*, which is a set of principles the observer should see effects of the confidential value for. These views of the program's execution are characterized by what is described as the *projection property*, which simply means that each view acts as a function that *maps* all faceted values to a corresponding non-faceted value for that specific execution. This projection property is what enables the authors to easily prove that faceted evaluation guarantees *TINI*. Each faceted value can be based on one or multiple principles in the form of nested faceted values resulting in a tree structure where the leaves are values, and the paths correspond to different possible views.

The big-step faceted evaluation semantics that are described, provide the rules for how a program with faceted values should be evaluated. The main intuition of faceted evaluation is that execution of the program "branches" only when a faceted value has influenced a previous computation. For instance, if there is a conditional statement which depends on a faceted value, this would be a point in the program where execution would branch, and both the `true` and the `false` statements would need to be fully evaluated simultaneously. There is also an auxiliary set of principles called the *program counter* pc, which keeps track of which faceted values the current execution branch is dependent on. The main principle that the faceted value in the conditional is dependent on gets appended to the pc. Principle k is added in the pc of the execution that uses V_H and principle $\bar{k}$ for the one dependent on V_L. And so value V_H is used in the private execution branch and V_L in the public one.

This faceted evaluation technique is proven to be *sound* and to guarantee *TINI*. It is also significantly *more performant* than the *shadow execution* or *secure multi-execution* approaches. The main difference, and reason for this performance gap is that faceted evaluation was designed to *only execute* "two instances" of the program when it is absolutely necessary. Faceted evaluation does not actually spawn completely different processes that run in parallel, but rather works by simultaneously evaluating multiple execution paths using their appropriate facets (values associated with their view of the system) when needed.

Another important aspect of providing guarantees that can see real-world usage is the fact that as much as non-interference is a strong guarantee that is usually desired, it is also sometimes overwhelmingly strong, and sometimes needs to be relaxed in real-world settings. The concept of relaxing this guarantee is coupled with the idea of having full *control* over the amount of private information that is disclosed to the public, when that (usually small) amount of information needs to be disclosed, that is possible without compromising parts of the data that need to remain hidden. This idea is known as *declassification*, which is intentionally releasing controlled portions of the private data to public observers. The most classic example of where declassification is needed is password checking, where even though a nugget of information about the password may be disclosed, it is still considered secure. Yet another up-side to faceted evaluation is that since the private and public facet are grouped together within the faceted value, this simplifies declassification since it allows for migrating information from one facet to the other by making small changes to the facets themselves.

4. Evaluation of the Faceted Values Approach

As discussed in previous sections, faceted evaluation provides a very good set of advantages over other methodologies. It does not need to resort to executing two separate copies of a program and thus *performs a lot better* than the alternatives. It also provides a *sound* approach and its faceted evaluation semantics and capabilities for doing *declassification* in a more intuitive and straightforward fashion.

Despite of all these attractive features and guarantees that techniques like faceted evaluation or secure multi-execution are able to provide they still suffer from the potential limitations of the fact that they only provide *termination insensitive* non-interference. As briefly mentioned, this slightly more relaxed variant of non-interference provides the guarantee that changes in private values will in no way alter public values; without however guaranteeing anything about the *time* that each branching process (or execution branch) may take to *terminate*. A simple example to demonstrate this can be seen in Figure **??**. In the example demonstrated in [2], there is a for-loop, the number of iterations of which will be executed do *not* on a private (or faceted) value. Even though this program will be accepted and no problematic flows will be detected, this can clearly disclose partial information about the secret value, based on the *termination channel* of the execution. Essentially the main caveat is that these techniques will allow iteration loops the *termination* of which may somehow depend on the value of a secret.

The work of Askarov et al. [2] is based on analyzing these cases where TINI, although a seemingly harmless compromise, could potentially disclose a relatively large amount of secrets. In their attempts to quantify the "amount" of information that can be disclosed through the termination channel. By making some necessary assumptions about the uniformity of the distribution of secrets, it is shown that the information gained by an attacker who has observed a polynomial amount of output is negligible in relation to the total size of the secret data. The overall conclusion is that even though TINI does seem to theoretically leak a large amount of information, the *best* an attacker can do is a *brute force* attack. This essentially means that it is technically *impossible* for the attacker to observe the secret value in *polynomial* time in comparison to the size of the secret data they are attempting to learn. Semantics for *termination sensitive* [6] non-interference are also proposed. In order however to prevent termination attacks, the type system resorts to necessarily disallowing loops dependent on confidential values, as well as adding extra requirements for there to be no loops in any of the branches of conditionals dependent on confidential values.

The above results therefore appear to provide a promising indication that guarantees like TINI provide for significantly stronger security guarantees than were initially expected. Although still vulnerable to them, the above work shows that information is not easily gathered from these potential leaks.

5. Conclusion and Future Challenges

In this short survey paper, we have mentioned a variety of techniques in the literature that allow for controlling information flow towards ensuring and maintaining certain security guarantees for sensitive data inside of the program. These techniques attempt to either statically or dynamically detect and track portions of the code that display problematic explicit or implicit information flows. Other attempts aim to completely bypass the issue by using two separate executions of the same program, one for each security class. That way secret information is never passed on to public observers, because the program execution potentially malicious observers can see does usually not even have handle these secret values at all.

There is however a slew of other channels that provide leakage points. There has been work on trying to control most of these channels, however such work usually means adding extra limitations and constraining the developer in terms of the way they need to write their code. The decoupled nature of these channels has also spawned equally decentralized research in these topics. The distinct characteristics of each of these side channels makes them very difficult to reason about in a unified fashion.

Of these covert channels, the *timing* and *termination* (which falls under timing) channels are the most vulnerable and exploited ones. There has been work towards providing both *time-sensitive* and by expansion *termination sensitive* non-interference [6, 12]. There have been proposed approaches that detect timing channel flows by using a type system [1], as well as using abstract interpretation to guarantee *abstract non interference* [10, 11] – another weaker variant of non-interference which works by modeling the *attacker* as an abstract interpretation whose task is to infer information about secret data from public data; All in the context of a specific attacker.

Interestingly enough, on the off hand, the existence of these language based techniques make it easier for malicious users to detect problematic programs that they can exploit and gather leaked information from! This is our own observation which is naturally not a concert for these systems, but should surely motivate the proper spread of awareness of these issues towards more robust and security checking.

The faceted evaluation approach is one which, while naturally doesn't provide a unified solution to all problems, provides a consistently secure, sound, and efficient solution for preventing unwanted information flow in programs within dynamic settings such as JavaScript and the web. While having been inspired by a wide variety of past approaches, it provides an entirely languange-based approach that simultaneously deals with explicit and implicit flows as well as simplifies declassification without introducing any apparent new limitations. After studying the related work as discussed in this survey, we conclude that language-based approaches such as this one are a significant step in the right direction towards a unified solution to providing the right security guarantees to users, when they are needed, by introducing the lease amount of overhead, complexity, and limitation as possible, and we believe that faceted values have, succeeded in doing that and have done so in the incredibly dynamic environment of JavaScript.

References

[1] J. Agat. Transforming out timing leaks. In *Proceedings of the 27th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 40–53. ACM, 2000.

[2] A. Askarov, S. Hunt, A. Sabelfeld, and D. Sands. Termination-insensitive noninterference leaks more than just a bit. In *Computer Security-ESORICS 2008*, pages 333–348. Springer, 2008.

[3] T. H. Austin and C. Flanagan. Efficient purely-dynamic information flow analysis. *ACM Sigplan Notices*, 44(8):20–31, 2009.

[4] T. H. Austin and C. Flanagan. Permissive dynamic information flow analysis. In *Proceedings of the 5th ACM SIGPLAN Workshop on Programming Languages and Analysis for Security*, page 3. ACM, 2010.

[5] T. H. Austin and C. Flanagan. Multiple facets for dynamic information flow. *ACM SIGPLAN Notices*, 47(1):165–178, 2012.

[6] G. Barthe, T. Rezk, and M. Warnier. Preventing timing leaks through transactional branching instructions. *Electronic Notes in Theoretical Computer Science*, 153(2):33–55, 2006.

[7] R. Capizzi, A. Longo, V. Venkatakrishnan, et al. Preventing information leaks through shadow executions. In *Computer Security Applications Conference, 2008. ACSAC 2008. Annual*, pages 322–331. IEEE, 2008.

[8] D. E. Denning. A lattice model of secure information flow. *Communications of the ACM*, 19(5):236–243, 1976.

[9] D. Devriese and F. Piessens. Noninterference through secure multi-execution. In *Security and Privacy (SP), 2010 IEEE Symposium on*, pages 109–124. IEEE, 2010.

[10] R. Giacobazzi and I. Mastroeni. Abstract non-interference: Parameterizing non-interference by abstract interpretation. *ACM SIGPLAN Notices*, 39(1):186–197, 2004.

[11] R. Giacobazzi and I. Mastroeni. Timed abstract non-interference. In *Formal Modeling and Analysis of Timed Systems*, pages 289–303. Springer, 2005.

[12] V. Kashyap, B. Wiedermann, and B. Hardekopf. Timing-and termination-sensitive secure information flow: Exploring a new approach. In *Security and Privacy (SP), 2011 IEEE Symposium on*, pages 413–428. IEEE, 2011.

[13] P. Kocher, J. Jaffe, and B. Jun. Differential power analysis. In *Advances in CryptologyCRYPTO99*, pages 388–397. Springer, 1999.

[14] A. Sabelfeld and A. C. Myers. Language-based information-flow security. *Selected Areas in Communications, IEEE Journal on*, 21(1):5–19, 2003.

[15] Wikipedia. Information flow (information theory) — Wikipedia, the free encyclopedia, 2015. [Online; accessed 13-December-2015].

[16] Y. Yarom and K. E. Falkner. Flush+ reload: a high resolution, low noise, l3 cache side-channel attack. *IACR Cryptology ePrint Archive*, 2013:448, 2013.

[17] S. A. Zdancewic. *Programming languages for information security*. PhD thesis, Cornell University, 2002.